Leonidas Boutsikaris

Management of complex data
14 March 2020

1st set of exercises

Question 1.1

For every movie that has more than one directors find the name of the movie and the director id's.

The field $primaryTitle$ is located in the title.basics file and the field $directors$ in the title.crew file. In the same way the operator IN is used in the SQL language, we can find out if the field of the $directors$ contains a comma and therefore multiple directors on a selected film.

1. $A \leftarrow title.crew$

2. $B \leftarrow title.basics$

3. $directors \leftarrow \sigma_{"," \ IN \ C.directors}(A)$

4. $movies \leftarrow (directors) \bowtie_{directors.const = B.const} (B)$

5. $\pi_{primaryTitle, directors}(movies)$

The question does not mention that is prohibited to re-format the title.crew file so we can change the file format by having only one director for each movie thus containing duplicates of the movies but single entries. For example:

tconst	director	writers
1	A	..
2	B	..
1	C	..

Now the relational algebra expressions will be

1. $A1 \leftarrow title.crew$

2. $A2 \leftarrow title.crew$

3. $B \leftarrow title.basics$

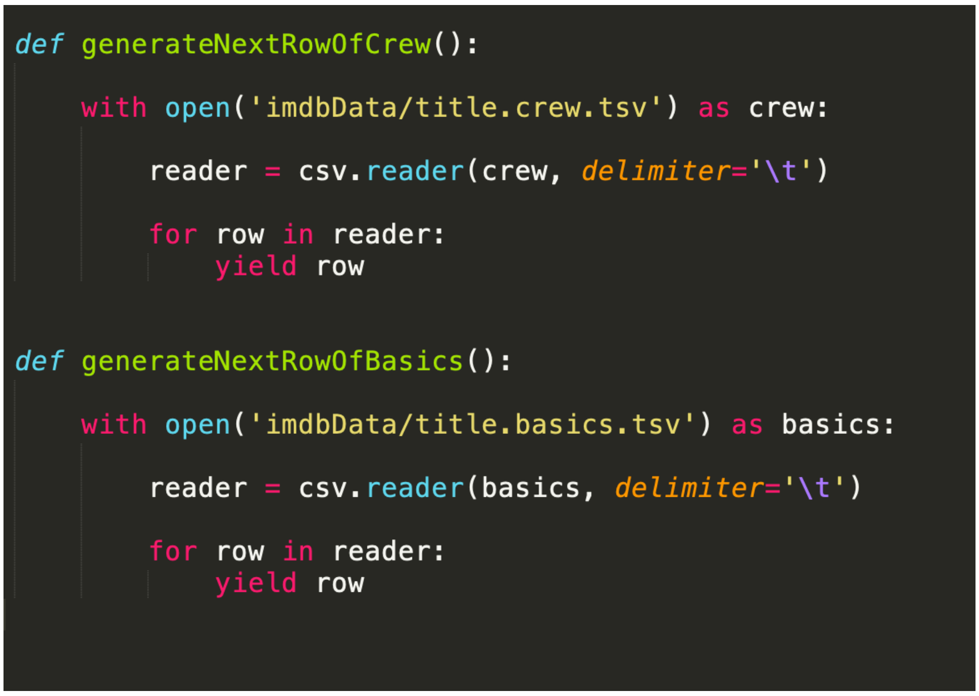
4. $directors \leftarrow \sigma_{tconst,directors}(A1 \bowtie_{A1.tconst=A2.tconst \wedge A1.director \neq A2.director} A2)$

5. $movies \leftarrow \Pi_{primaryTitle}(directors \bowtie_{directors.const=B.const} B)$

Even though we cannot get the directors in the example output format we can obtain the list of movies that have more than one director. This is a more “relational algebraic way” to obtain them rather than using the IN operator above.

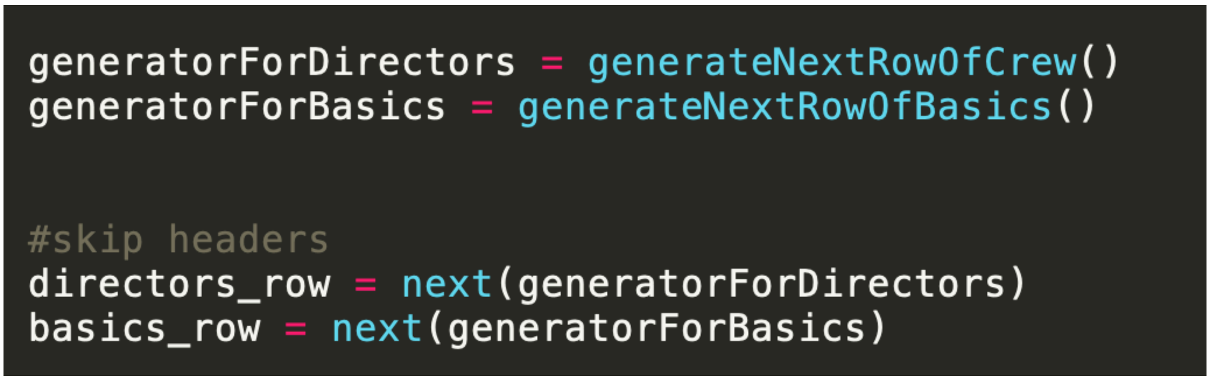
Q1.1 Programming

The files are not allowed to be store in memory so they must gradually be accessed. This is achieved by using the ***yield*** keyword. The ***yield*** keyword suspends function’s execution and sends a value back to the caller, but retains enough state to enable function to resume where it is left off. When resumed, the function continues execution immediately after the last yield run. This allows its code to produce a series of values over time, rather than computing them at once and sending them back like a list. Bellow we can see the functions that implement it.



```
def generateNextRowOfCrew():

    with open('imdbData/title.crew.tsv') as crew:

        reader = csv.reader(crew, delimiter='\t')

        for row in reader:
            yield row

def generateNextRowOfBasics():

    with open('imdbData/title.basics.tsv') as basics:

        reader = csv.reader(basics, delimiter='\t')

        for row in reader:
            yield row
```

Yield is used in Python generators. A generator function is defined like a normal function, but whenever it needs to generate a value, it does so with the ***yield*** keyword rather than return. If the body of a def contains ***yield***, the function automatically becomes a generator function. Each time we call the next() function we get a new row of our file.



```
generatorForDirectors = generateNextRowOfCrew()
generatorForBasics = generateNextRowOfBasics()


#skip headers
directors_row = next(generatorForDirectors)
basics_row = next(generatorForBasics)
```

This method will be used on most of the questions so there is no need to re-explain it. The only thing that changes are the conditions we want according to the question.

Question 1.2

For every title that is the first episode of a series we have to find the name of the title, the id of the series and the season number.

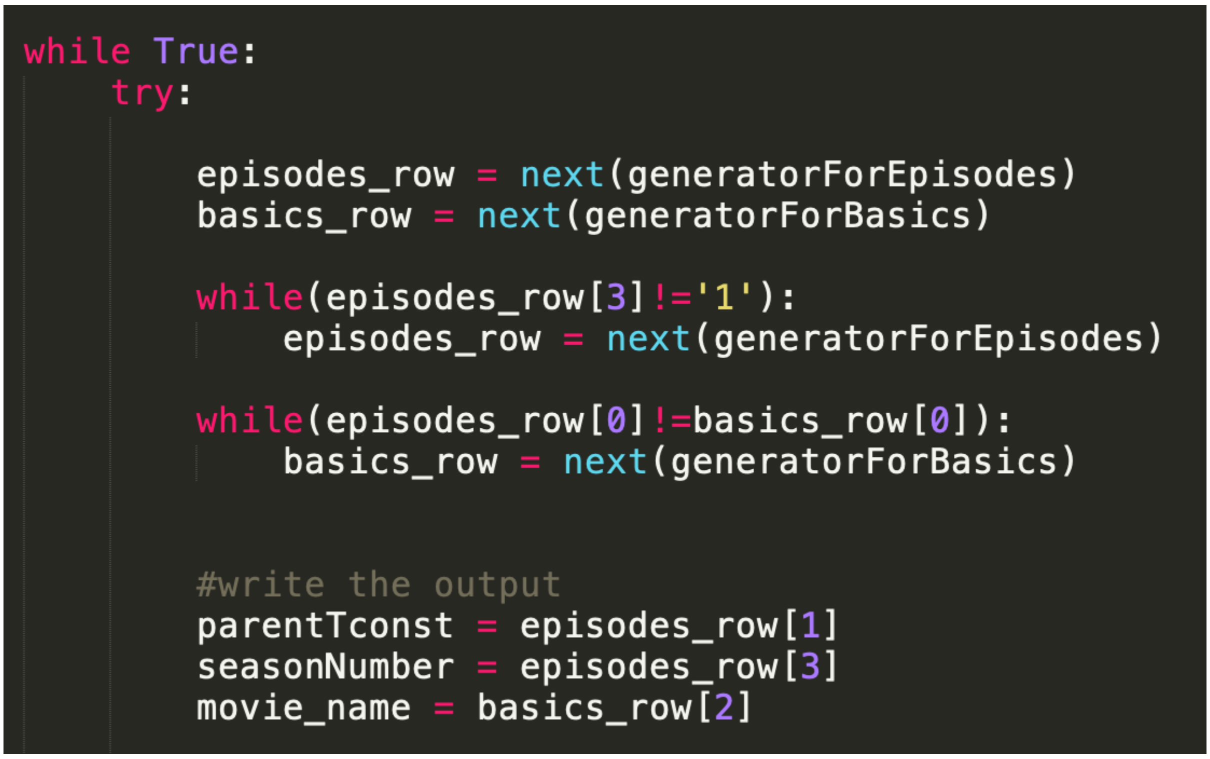
1. $A \leftarrow title.episode$

2. $B \leftarrow title.basics$

3. $x \leftarrow \Pi_{PrimaryTitle,parentTconst,seasonNumber}(A \bowtie_{A.parentTconst=B.tconst \wedge B.episodeNumber=1} B)$

Q1.2 Programming

We know as a fact that the files are sorted. The basics file has all the titles while the episode file only has the titles that are an episode of a series, not a movie. Thus we can cross check with a simple merge algorithm. The ***yield*** logic also applies here.



```python
while True:
    try:

        episodes_row = next(generatorForEpisodes)
        basics_row = next(generatorForBasics)

        while(episodes_row[3]!='1'):
            episodes_row = next(generatorForEpisodes)

        while(episodes_row[0]!=basics_row[0]):
            basics_row = next(generatorForBasics)


        #write the output
        parentTconst = episodes_row[1]
        seasonNumber = episodes_row[3]
        movie_name = basics_row[2]
```

The outer while refers to the EOF.

tconst of BASICS	parentTconst	seasonNumber	..
1	2	2	..
2	3	1	..
3	7	1	..
4	10	..	..
5	..	..	..
6	..	..	..
7	..	..	..
..	..	..	..
N	N	..	..

The first inside while skips rows until it finds an episode with a season number of 1(red arrow). The second while skips rows until it finds a matching tconst on the basics file(green arrow). As mentioned above we are sure this can be done because the basics file contains every title.

Question 1.3

Find all the title names that don't have a rating.

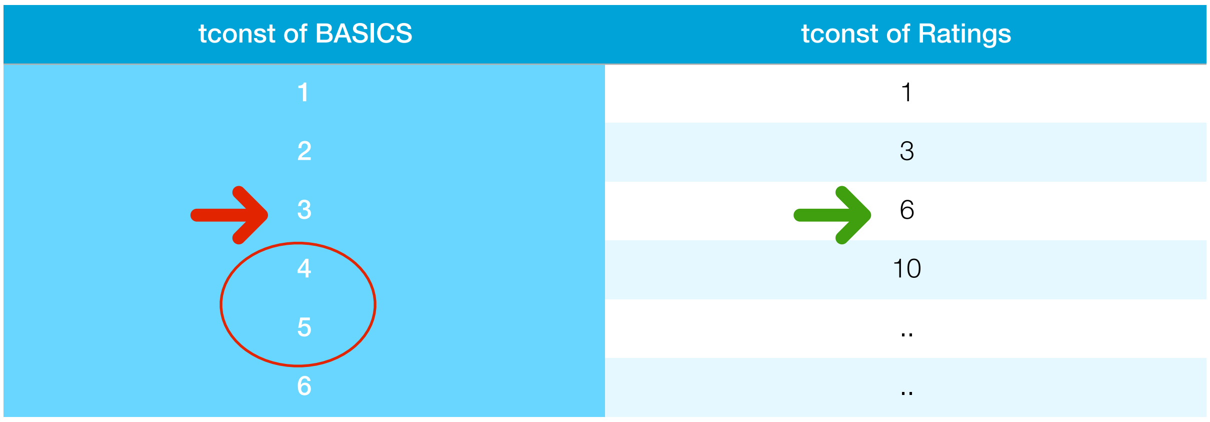
1. $A \leftarrow title.basics$

2. $B \leftarrow title.ratings$

3. $Movies \leftarrow \Pi_{PrimaryTitle}(A \bowtie_{A.tconst=B.tconst} B)$

Q1.3 Programming

The same logic applies with Question 1.2. Basics file have all the title while ratings have missing ones. A simple algorithm checks if the tconst ids are matching. If the id **does not** match we print the movie name to the terminal output from the basics file. If the id matches we skip to the next ratings row. In each case the pointer on basics (Red arrow) will iterate. For example red arrow is at tconst=1, green at ratings tconst=1. Green arrow moves because the ids match and is now at tconst=3. Red arrow also moves. Now the ids don't match so we will have an output. If the id's matched the same thing will happen so we are covered. Essentially we are saying that if the tconst pointer of the ratings is bigger than the tconst of basics print all the missing movies until we reach the tconst of the ratings and then check again. For example, when the green arrow reaches tconst=6 in ratings the red arrow will print the movie 4 and 5 until it reaches tconst=6 in basics.



tconst of BASICS	tconst of Ratings
1	1
2	3
3	6
4	10
5	..
6	..

Question 2.1

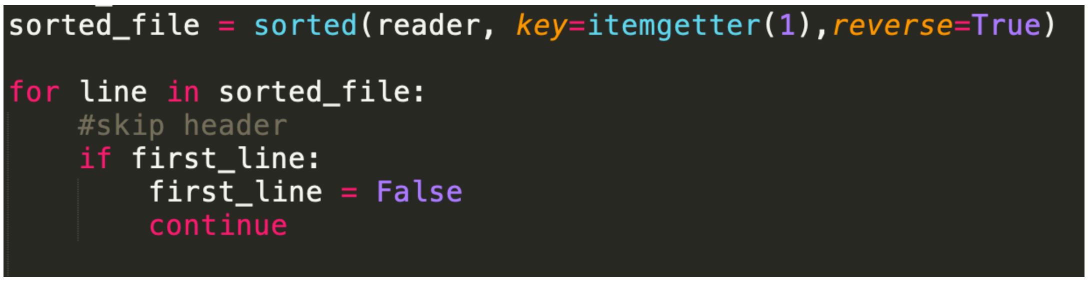
Movies are rated with a 10 star system and intervals of 1 star. For example 0.1 to 1, 1.1 to 2 and so on. Find out how many movies lie in each interval with a sorting and a hashing method.

1. $A \leftarrow title\,.\,ratings$
2. $firstInterval \leftarrow \Pi_{tconst}(\sigma_{averageRating>0.1\ \wedge\ averageRating<1}A)$
3. $G_{count(tconst)}\ firstInterval$

Now repeat for every Interval.

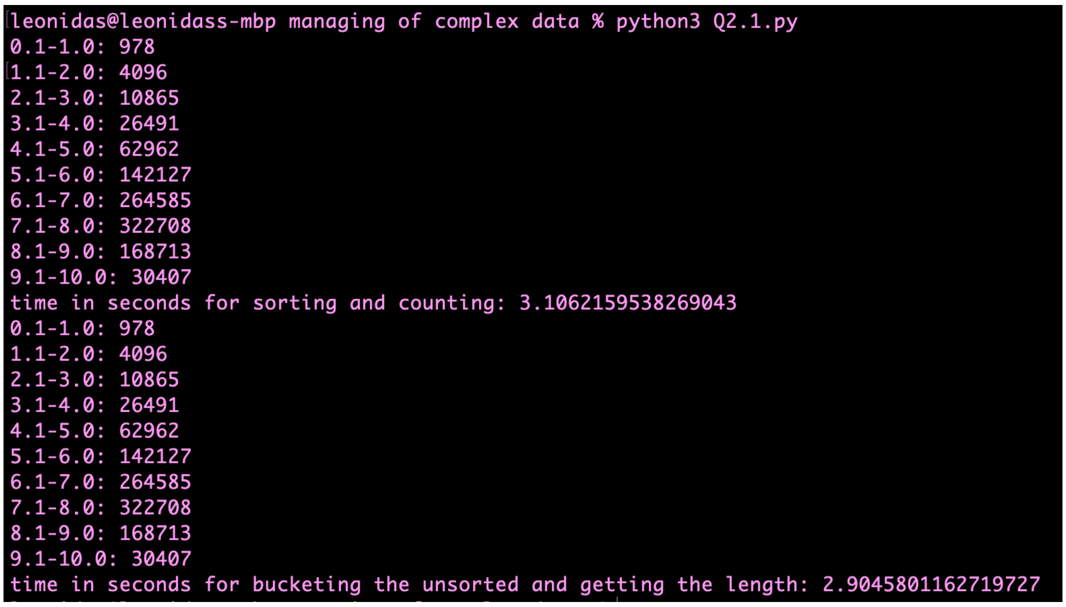
Q2.1 Programming

For the sorting method we will first sort the tsp file according to the column we are interested using the itemgetter(1) for the second columns as a sorting key. Then inside the for loop we will count every occurrence. Then we will just print the results.



```
sorted_file = sorted(reader, key=itemgetter(1),reverse=True)

for line in sorted_file:
    #skip header
    if first_line:
        first_line = False
        continue
```

For the hashing method we will iterate in the unsorted file and put every movie inside one of the 10 buckets for each interval from 1 to 10. Then we have a simple operation of getting the length of each bucket. Bellow we can see the results for both methods and observe that the hashing method is doing better in terms of seconds.



```
leonidas@leonidass-mbp managing of complex data % python3 Q2.1.py
0.1-1.0: 978
1.1-2.0: 4096
2.1-3.0: 10865
3.1-4.0: 26491
4.1-5.0: 62962
5.1-6.0: 142127
6.1-7.0: 264585
7.1-8.0: 322708
8.1-9.0: 168713
9.1-10.0: 30407
time in seconds for sorting and counting: 3.1062159538269043
0.1-1.0: 978
1.1-2.0: 4096
2.1-3.0: 10865
3.1-4.0: 26491
4.1-5.0: 62962
5.1-6.0: 142127
6.1-7.0: 264585
7.1-8.0: 322708
8.1-9.0: 168713
9.1-10.0: 30407
time in seconds for bucketing the unsorted and getting the length: 2.9045801162719727
```

Question 2.2

For every starting year you have to find the average of the ratings of this year.

1. $A \leftarrow title.basics$

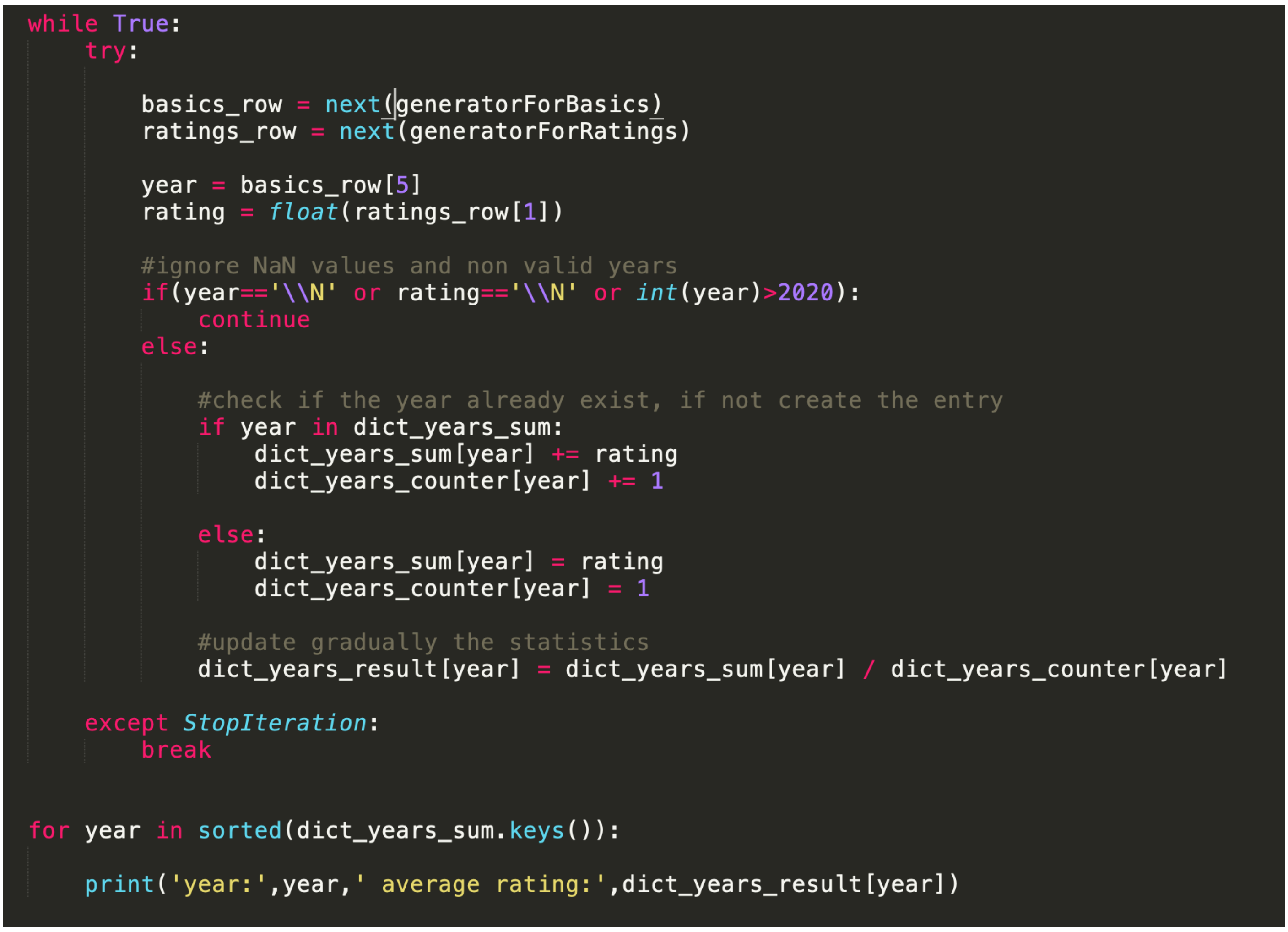
2. $B \leftarrow title.ratings$

3. $Averages \leftarrow \Pi_{year,averageRating}(A \bowtie_{A.tconst=B.tconst} B)$

4. $year\ G_{avg(averageRating)}\ Averages$

Q2.2 Programming

The ***yield*** and generators logic is also used here. Three dictionaries are used that gradually count everything for each year. The sum of ratings until now, the number of ratings and the result that these two metrics gradually produce.



```python
while True:
    try:

        basics_row = next(generatorForBasics)
        ratings_row = next(generatorForRatings)

        year = basics_row[5]
        rating = float(ratings_row[1])

        #ignore NaN values and non valid years
        if(year=='\\N' or rating=='\\N' or int(year)>2020):
            continue
        else:

            #check if the year already exist, if not create the entry
            if year in dict_years_sum:
                dict_years_sum[year] += rating
                dict_years_counter[year] += 1

            else:
                dict_years_sum[year] = rating
                dict_years_counter[year] = 1

            #update gradually the statistics
            dict_years_result[year] = dict_years_sum[year] / dict_years_counter[year]

    except StopIteration:
        break


for year in sorted(dict_years_sum.keys()):

    print('year:',year,' average rating:',dict_years_result[year])
```